Performance and Analysis of Multithreaded Systolic Array

Nir Shaul and Manor Zvi

***Abstract*— As Moores law slows down (and soon expected to completely stop), Power and utilization become a limiting factor towards keep improving our computing systems. Many architects nowadays believe that major improvement in cost-energy performance must now come from domain-specific hardware. One of the most popular domains today is the area of Machine Learning calculations, and to be more specific Deep Neural Networks (DNN). Numerous DNN accelerators use a Systolic Array to address the common computation intensity in DNN workload in a batch environment. The Systolic Array has the advantage of a simple I/O interface and a confining structure without control. Another advantage of the Systolic Array lays in its data flow structure. As will explained later, data pass through multiple computation units, before being returned to memory. As the same data are used repeatedly, the computation throughput increases, without a need to for increasing the I/O interface or using local memory. However, Systolic Array is a rigid structure that is NOT efficient in a sparse (with zeros) environment. The Multithread Systolic Array (MTSA) (patented by Huawei) target is to resolve the inefficiency caused by data and weight sparsity.**

I. INTRODUCTION

DSP, Image Processing as well as linear algebra algorithms have found an efficient implementation medium in parallel computing domain. These algorithms can be efficiently mapped onto systolic arrays, due to their regularity on the data level. Systolic arrays were designed to tackle fine-grained computation and exploit data level parallelism of these problems directly. They feature a very homogeneous design and consist of several identical processing elements (PEs).

As the need to improve Machine Learning computations increases, most of today silicon vendors are developing AI chips of their own. The most known is Googles Tensor Processing Unit (TPU). In their paper from 2015, they describe an architecture based on single-threaded 256X256 8-bit Systolic Array, that can achieve 15X-30X average speedup over contemporary CPU or GPU, and roughly 80X TOPS/Watt [1]. In their paper, Google admitted that for some applications, the TPU failed to achieve high utilization, a problem that we aim to solve.

Another major player in the field of Machine Learning is Facebook. In their paper from 2018, they describe their data center infrastructure for machine learning training and inference [5]. From a hardware point of view, Facebook guys choose to utilize a huge fleet of CPUs and GPUs in order to support the training frequencies at the required service latency. For Machine Learning inference, Facebook currently relays mostly on CPUs for all major services, though they constantly prototype new hardware approaches for Machine Learning Tasks.

Many more key players in the field of cloud and distributed computing are working hard now to have a piece of the cake: Amazon is building an ecosystem of cloud infrastructure by AWS (Amazon Web Services) [6], Intel acquired Nervana to provide DNN accelerators (inference and training) and many more.

Our paper will focus on performance analysis of the Systolic Array which combines the concept of multithreading calculation in order to reduce the inefficiency caused by the sparsity of the input matrices. In this paper, we will describe the basics of Systolic Array computation for matrix multiplication (1), then focus on the elements of MTSA concept and its HW implementation properties (2), then focus on the properties of the MTSA simulator we have developed (3), and lastly review the results and analysis compared to the traditional single-threaded Systolic Array architecture.

In the literature, there are many variations of traditional non-multithreaded Systolic Array architectures for matrix multiplication. These variations focus on three subjects: how the data streams across the PE's, how the results are collected, and whether the architecture treats differently to different streams of data (in DNN: data and weights). In figures 1 and 2 we will discuss some examples for these different architectures, in order to understand these variations before we focus on the one we chose.

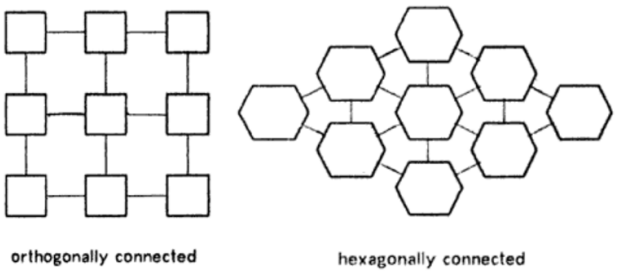

Fig. 1: **Hexagonally connection topology versus orthogonal connection topology for systolic arrays architecture.** The topology dictates the data flow through the system. This variation in connectivity is generated from differences in how data streams across the PE's. In the hexagonal connection topology, the result of each multiplication streams between the PE's as well as the data and the weights. In the orthogonal connection topology only data pass (1 or 2 streams), and the intermediate result is being saved in each PE and weights streams as will be explained next.

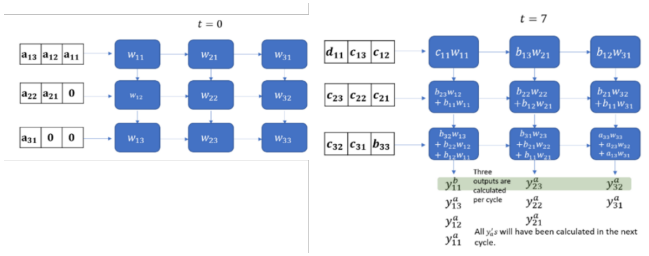

Fig. 2: **: Example of data streams across static pre-loaded weights, in Orthogonal Systolic Array.** . In this figure, you can notice how the data flows through the system. This topology is non-symmetric with respect to different streams of data. The weights need to be pre-loaded to the PEs at a preliminary phase before the calculation starts. This variation is the one which is implemented in Google's TPU.

II. Operation concept of the chosen Systolic Array architecture

In our analysis, we chose to focus on symmetric (NxN) Systolic Array topology which its inputs matrixes (data matrix and weights matrix) are treated equally by the architecture (no preload data). The key advantages of this topology lay in its simplicity and robustness for different workloads: different DNN algorithm architectures might need to frequently reload the weights, for example in case of very wide layers or convolution layers. The ability to fetch weights simultaneously with the related data might be useful. The reason for that choice is that this is in our opinion the most generic topology, but from the other hand, the most simple to understand and to implement in hardware.

The data matrix and the weights matrix are passing through the PEs like a wave front. Every PE calculates the product of its inputs, add it to the previous intermediate result, and save the result to a local register. Let N be the input matrices edge (NxX matrices), after N calculations, the result accumulated in the PE is the result of the computation, while the location of the PE in the array is equivalent to its algebraic position in the result matrix, as shown in figure 3.

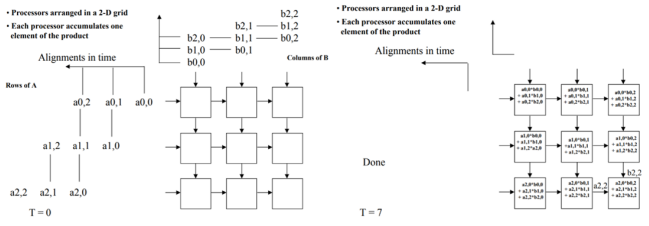

Fig. 3: **: Traditional architecture of single-threaded and symmetric Systolic Array for matrix multiplication.[4]** Data input are passing from west to east while weights passing from north to south. The result collected in every PE. In order to synchronize the array operation, data rows and weights columns are being fetched in consecutive clock cycles: on CC 1: data [0,0] and weight [0,0], on CC 2: data [0,1], data [1,0] and weight [1,0], weight [0,1] and so on. In this example, all the results are ready after 7 cycles.

An important observation here is the fact that in this architecture, each PE computes the product of its inputs, no matter if they are zeroes either non-zeroes integers. At every clock cycle, each PE see 2 numbers in its input ports. If one of them or both are zero, the result has already known, and the controller can turn down clock enable of the MAC to save power. in such case, the inputs just bypass the MAC and are being saved to each PE output register port. When it comes to sparse matrices, the system could be much more efficient if the MACs could work on something else, when one pair of inputs detected as zeroes/half zeroes. It turns out that, in most DNN software architectures this is a real problem (see table 1).

Regards single threaded Systolic Array, there is almost no hardware overhead (over the PEs basic elements): The inputs are being fetched using large local FIFO buffer (see Googles TPU paper Unified buffer), that is mandatory in our architecture also in order to synchronize the system with its host CPU. There are no buffers between PEs to synchronize PEs with each other and very small control (only 2% of the die, in the TPU case).

TABLE I: LeNet weight and activation sparsity after training with L1 and L2 regularization. The relative test accuracies are L1 99.0% and L2 99.8% [5] Sparsity measured as a percentage of zero elements in a matrix. As you can see, the percentage of zero weights is greater than 50% in some cases, and the activations are not much better.

Layer	L2 wights	L1 wights	L2 activations	L2 activations
conv1	0.142	0.289	0.717	0.662
conv2	0.491	0.505	0.528	0.662
fc3	0.258	0.502	0.661	0.593
fc4	0.258	0.504	0.000	0.000

In the following parts, we will present our multithreaded Systolic Array approach, which basically aims to keep all the MACs busy during the system operation.

III. Multithreaded Systolic array concept for HW implementation

The main goal of the MTSA is to ensure that the multipliers which are in every PE, work as much as possible especially in case of sparsity. The concept is to feed each PE with enough parallel data streams, so in case of zero-detection in an input port, the multiplier could execute the calculation of other thread (other matrix multiplication) instead of just bypass zero. Characteristics of the MTSA architecture:

- The inputs are M matrices of data and M matrices of weights, while M is the multithreading order (number of parallel threads). In fact, one can relate the inputs as 3D tensors instead of two-dimension matrices, in single-threaded Systolic Array.

- Interconnect between each PE to its neighbors become vectored of order M as well.
- Every PE holds the current thread to be calculated the in the current thread register (CTR).
- Switch-On-Event policy in each PE, asynchronous to other PEs. The trigger is zero detection in an input port.
- Switching policy of Round Robin in each PE, in order to avoid starvation of one thread. As said, each PE holds the last thread id that has been calculated the last cycle. In the next cycle, the controller will start at that id, then iterate through all the threads, until he finds an input pair that both arent zeroes. Then that thread would be calculated, and its id would save into the CTR.
- On-Cycle context switch: In case of zero detection, the multiplier calculates the following thread in the same cycle, and update the current thread register.
- In each clock cycle, all zeroes are bypassing the MAC, and shift directly to next buffers.
- Interconnect implemented as M dimensional FIFO buffer, in order to handle asynchronously between PEs, caused by different CTR values in adjacent PEs

IV. SIMULATOR PROPERTIES

In order to analyze the proposed architecture, we developed an SW simulator of the MTSA HW. The Simulator has written in Python for its simplicity and extensive numeric and graphics support. The purpose was to design the simulator building blocks with high correlation to the HW components so the generic parameters would indicate the same parameters for the HW. The simulator also has features to extract statistics regards the operation.

The simulator SW structure:

- Systolic Array: Top Class of the Simulator. Contains three 2D lists (nested python lists), one for PEs (the Systolic array itself), one for horizontal buffers, and one for vertical buffers. In addition, for statistics extraction purposes, Systolic Array instance holds clock cycles counter and results and utilization per PE arrays. The class has a method to stimulate one clock positive edge step, in which it performs all the actions that real Systolic array would take.
- PE: is the basic compute element. It has pointers to all 4 adjacent buffers (west, north, east and south), a threads-size list to hold intermediate results and attribute onThread to symbolize the CTR (current thread register). It has a method to connect itself to adjacent buffers in the Systolic Array and another method to perform on clock cycle operation. The PE gets its inputs from the west and north buffers, save intermediate result inside and push outputs to east, south buffers. One class inherits from it:
  - o PElimited: Same as PE but support limited buffer size.
- Buffer: a data structure to hold the data between PEs in MTSA. It contains a Pythonic dictionary, each key is a thread id, and each value is a FIFO list of literals for the next PE. 2 classes inherit from it:
  - o FIFO: a special buffer for inputs to the systolic array
  - o OUTPUT: a special buffer for outputs from the systolic array.
- Logger: log all activities to a text file.

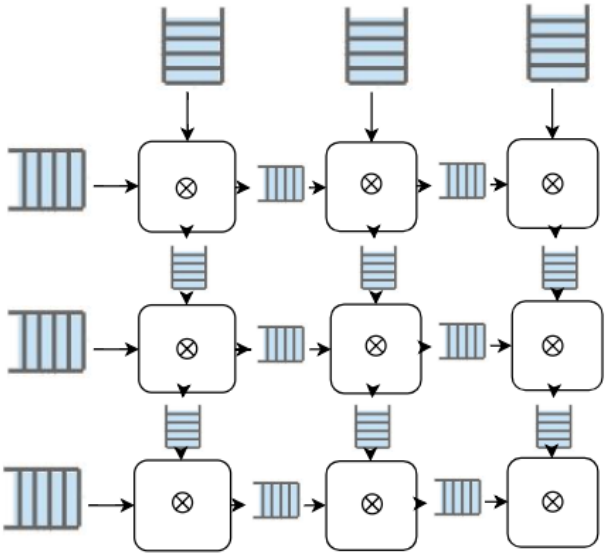

Fig. 5: **Schematic view of the components of the SW simulator of the MTSA.** The Systolic array holds all of the components of the array. Every PE is connected to its neighbors as reference attributes.

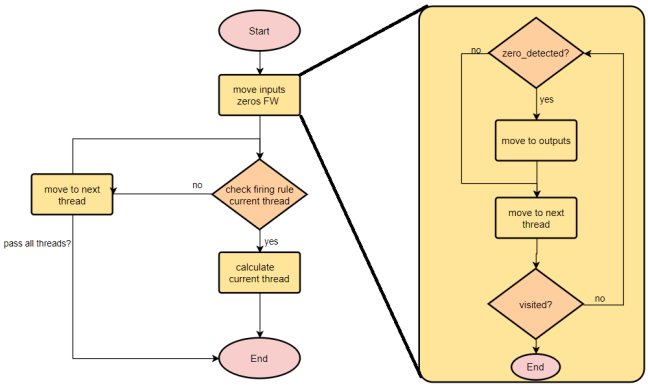

Fig. 6: **Decision Tree of the Calculation in the MTSA PE.** This decision tree occurs every cycle and include detecting zero, calculating next thread (context switch) and update the CTR in case context switch. The firing rule is to calculate when both inputs of current thread are not zeros.

V. RESULTS AND ANALYSIS

A. 1. Average utilization of the multipliers over sparsity density, with no buffer's depth limitation:

Figures 9, 8 present average multipliers utilization in each PE over the sparsity density. utilization calculated as

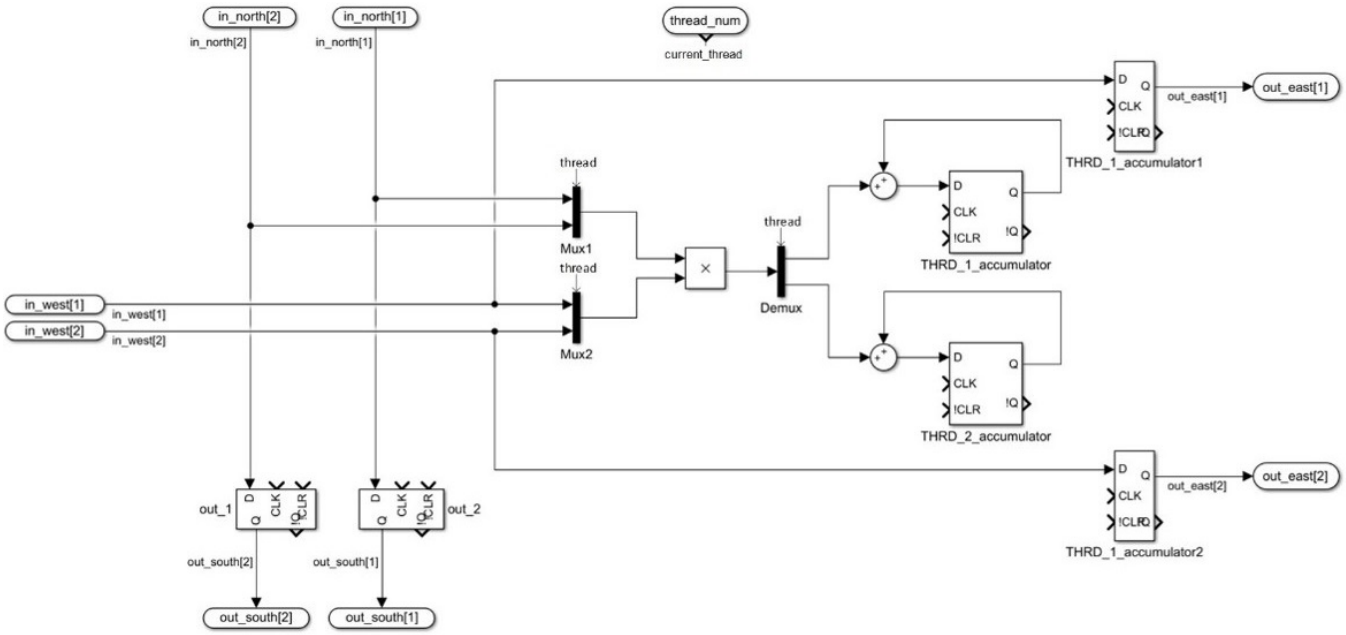

Fig. 4: **HW Description of PE in Multithread Systolic Array of 2 threads .** The differences from the traditional PE include: expanding the inputs and the outputs, expanding the accumulators, adding two multiplexers and one de-multiplexer, and current thread register. Note that the zero-detection logic does not appear in this image.

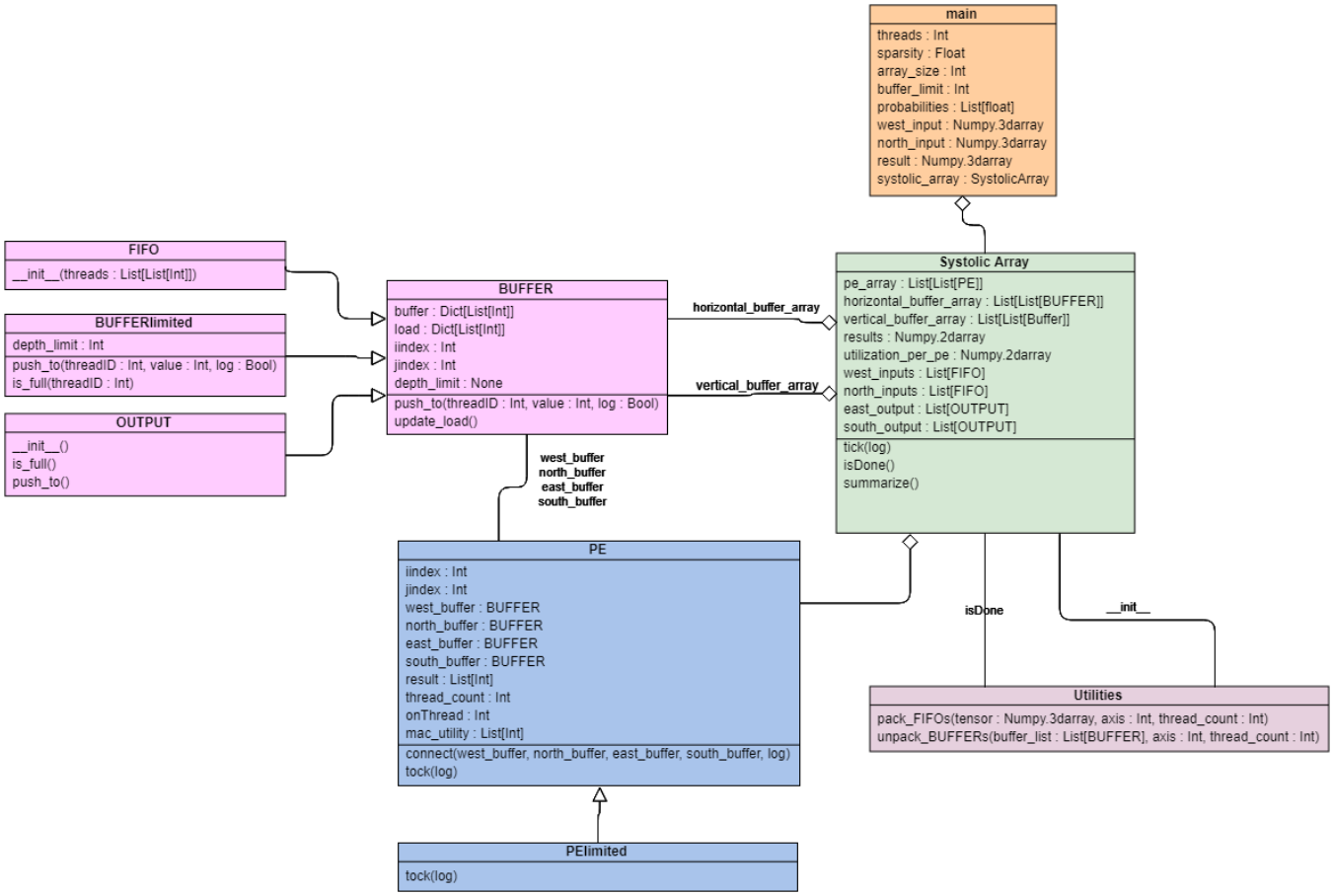

Fig. 7: **Class Diagram of the MTSA Simulator.** The simulator consists mainly of 3 functional classes (Systolic Array, PE, Buffer) and utilities.

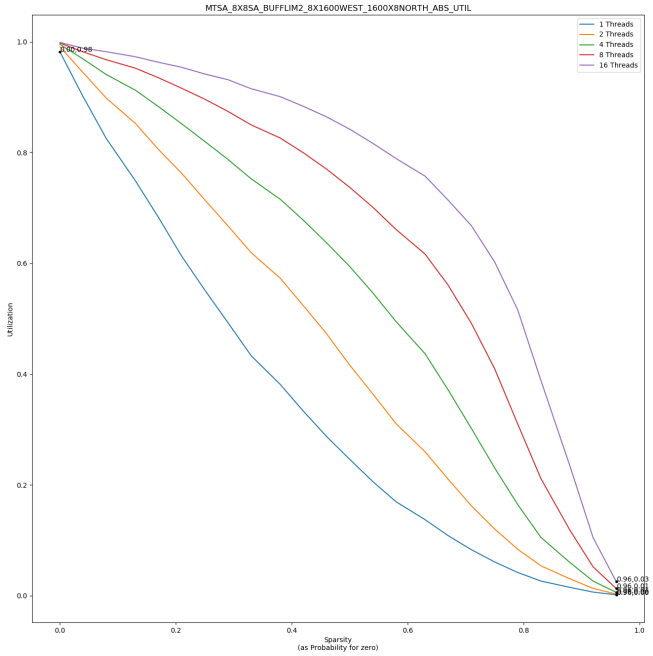

Fig. 8: **MTSA average Utilization Improvement over sparsity for a different number of threads.**

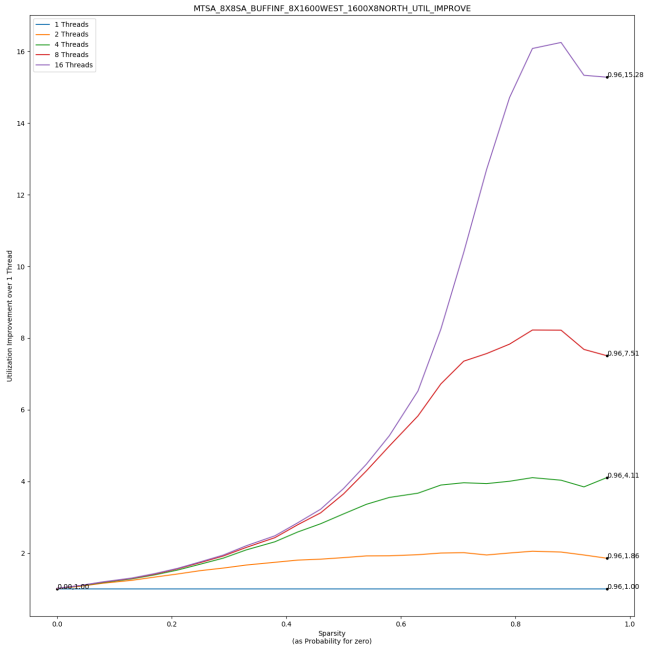

Fig. 9: **MTSA average utilization over sparsity for a different number of threads.**

the percentage of cycles in which the multiplier did work. Sparsity measured as the probability for zero at each cell of input matrices. The figure shows us the utilization as a function of the sparsity, for 1,2,4,8,16 threads, and the right present the Utilization Improvement over a single thread. We aimed to measure performances under steady-state of the MTSA, therefore we fed the system with very long matrices: 8X1600 from the west, and 1600X8 from the north. In addition, due to the stochastic nature of sparsity, we performed a few dozens of experiments and took mean over it to obtain an approximation for the real sparsity distribution over the matrix. Of course, much more experiments are needed to obtain high statistical significance.

Figure 8 fits our expectations, especially for sparsity values 0 and 1.

- You can see in the plot that for sparsity value 0, the utilization doesn't differ from thread number to another. Because there are no zeroes in the input matrices at all, all multipliers would perform work at each clock cycle with probability 1.0.
- For sparsity value very close to 1, you can see that the utilization converges to zero, as expected, from the same reason.

For any sparsity value, we expected the utilization follow the formalism in figure 10. You can see that for sparsity 0.5 as an example, for 1 thread we got utilization approximately equal to 0.27 (0.25 expected), and for 2 threads we got utilization approximately equal to 0.47 (0.4375 expected). result it can be clearly seen that as the number of threads increases, the utilization improvement is higher for any sparsity value as expected. Moreover, it can be noticed from the graph that for each threads number, there is sparsity value which is a "knee

$$P_{\substack{one\\thread}}(work|in_{west}, in_{north}) = P_{\substack{one\\thread}}(in_{west} \neq 0, in_{north} \neq 0) = (1-sparsity)^2$$

$$P_{\substack{two\\threads}}\left(work\middle|\begin{matrix}in1_w, in1_{n'}\\ in2_w, in2_n\end{matrix}\right) = P_{\substack{two\\threads}}\left(\begin{matrix}(in1_w, in1_n \neq 0 \ \cup in2_w, in2_n \in \mathbb{R}) \cap\\ (in2_w, in2_n \neq 0 \cup in1_w, in1_n \in \mathbb{R})\end{matrix}\right) =$$
$$= (1-sparsity)^2((1-sparsity)^2 + 2sparsity(1-sparsity) + sparsity^2) +$$
$$(1-sparsity)^2(2sparsity(1-sparsity) + sparsity^2)$$

Fig. 10: **multiplier probability to perform work for 1,2 threads. The probability correlates the utilization for large sample sizes.**

point" of the utilization improvement. That point shall be the optimal operation point for utilization versus HW overhead cost. In other words, for each sparsity value, one can define an architecture to obtain an optimal performance-hardware overhead ratio. As sparsity increases, the utilization of the MTSA gets low (as well as for the traditional MT), but it can be clearly seen from the result that it takes place in much higher values of the sparsity. For example, The MTSA 16-thread architecture keeps the high utilization until 60% of sparsity!

B. Average clock cycles for matrices stack completion over sparsity density, with no depth limitation on the buffers

Figures 11, 12 presents the amount of clock cycles, needed for 1 matrix-matrix multiplication on average. Time measured in clock cycles terms for complete a stack of matrices. Then averaged over the number of matrices in the stack to obtain the average clock cycles per matrix. Speed up calculated as the number of clock cycles per matrix on average, divided by a single thread cycles count.

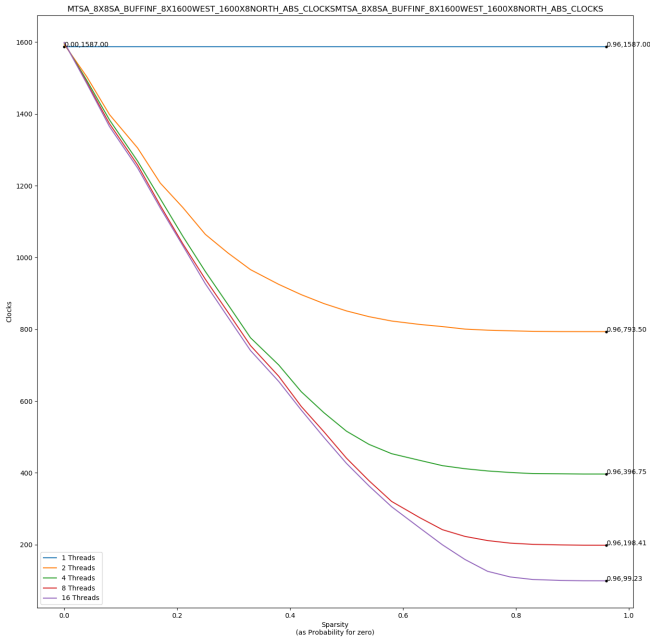

Fig. 11: **MTSA average clock cycles for completion of a data batch over sparsity, for different number of threads.** This measurement is in inverse relationship with the measurement of calculation throughput which is in terms of matrixes per second.

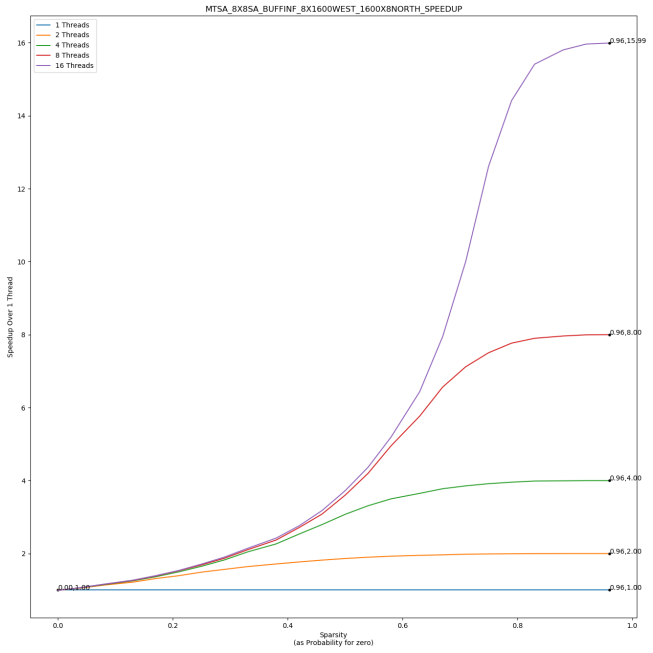

Fig. 12: **MTSA speedup over sparsity, for a different number of threads.**

Figure 11 emphasize the effect of the number of threads on the throughput. it can clearly see that the throughput of the system is higher for large amount of threads, and for higher sparsity. The MTSA system delivers its maximum throughput (minimum clock cycles for completion one batch) as sparsity get close to 100% which makes the system just pass the zeros simultaneously for all the threads.

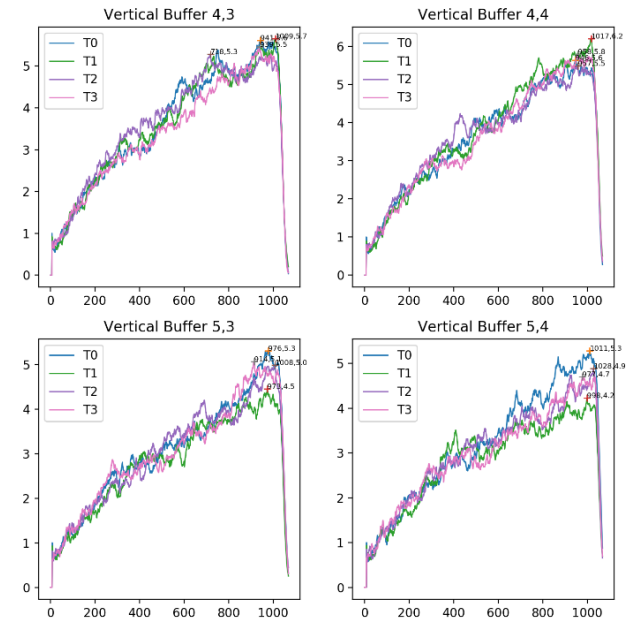

Fig. 13: **MTSA buffer size for central 4 PEs for 4 threads, over clock cycles, under sparsity of 0.5.** Note the monotonically nature of the graph and the maximum point achieved right before the drop. The cause for the drop is the inputs ending, and the fact that the maximas achieved right before taught us that for larger inputs it could keep growing.

This results emphasis the improvement of the MTSA architecture over the traditional SA. MTSA harvest the sparsity nature of the matrices to improve its throughput, while the traditional architecture delivers fixed throughput for any sparsity value. Another interesting feature to notice, is the correlation between Utilization Improvement to speedup (figures 12 and 9). the traditional SA increase matrix throughout in exchange to relatively low utilization (Google report in their paper that their design issued in some case utilization as low as 27%). The comparison above reveals that the utilization improvement and speed up behave the same.

C. Buffers effective depth over time

In order to analyze the optimal hardware overhead for different sparsity values, we drew the depth of each FIFO in every buffer, over time of operation. In order to lower the effect of sparsity spatial distribution over the input matrices on the results, we conducted the experiments few hundreds of times, and averaged over the results. The conclusion, however, wasnt too surprising: for a fixed sparsity value, each thread FIFO didnt manage to evacuate quick enough, and got bigger and bigger. It seems that there is no steady state for buffers depth, and their depth raises monotonically (see figure 13). Therefore, there is an obvious reason to analyze the MTSA performances under buffers depth limitations.

D. Performances over sparsity, for minimal buffer depth

Surprisingly, limiting the buffer depth to the minimal depth possible in the simulator (depth equal 2 only 2 literals per thread can wait between PEs), didnt change the performances much.

Moreover, the little change which did happen, took place only for small values of sparsity, and caused the graphs to recede from each other (compare figure 14-right and figure 12, and focused on the small sparsity values). However, the relation between speed up and utilization improvement didnt change, the speed up and utilization improvement are still highly correlated.

E. MTSA Performances over buffer depth limitations, for fixed sparsity value, and different number of threads

From reviewing figures 15 it seems that for relatively large values of sparsity, buffer depth almost doesnt affect the performances. That discovery is quite surprising, because we expected to have larger tension on the buffers, as the sparsity increase. As the sparsity increase, at each clock cycle, more literals want to pass to the next buffer (see figure 6) and accumulate in it. Therefore, we expected that the limitation would have more influence on the performances for larger values of sparsity. We denote that we currently cant explain the local maxima in the middle of the graph, which might indicate that buffer limitation of 6 would be optimal but make no sense. Although like always, we have conducted several experiments and averaged over them, much more trails are needed.

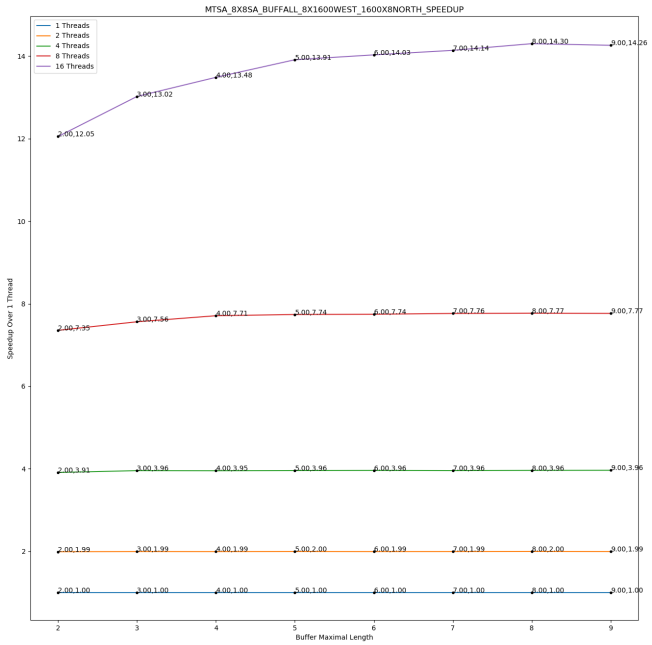

Fig. 16: **Speedup for fixed sparsity of 0.79, over different buffer depth limitations, for different number of threads.**

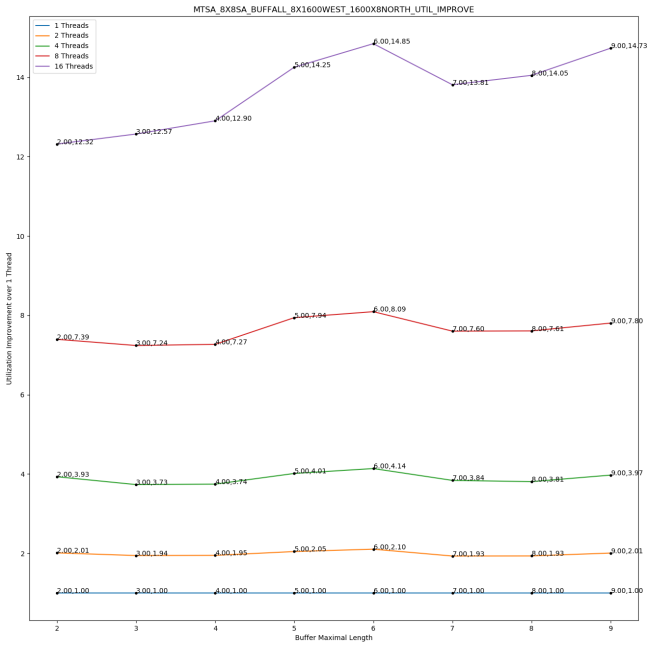

Fig. 17: **Utilization improvement for fixed sparsity of 0.79, over different buffer depth limitations, for different number of threads.** . Like always, we fed the system with long matrices: 8X1600 from the west, 1600X8 from the north.

VI. HW overhead Estimation over number of threads and matrix size

The HW overhead of MTSA over SA consist of the following:

- Overhead for every PE:
  - M accumulators
  - M counters
  - Two multiplexers of M inputs each
  - One de-multiplexer of M inputs
  - Zero Detecting logic
- Backpressure FIFO's which consist of M data cells (registers)

Let N be the order of the input matrixes and M be the order of the multithreading. Therefore, the number of PE's is N^2, and number of buffers is $2N(N-1)^2$. The traditional SA consist of the same number of Input - FIFO or equivalent, so they are not part of the overhead.

TABLE II: HW overhead Estimation over number of threads and matrix size

	MTSA	Traditional SA	Overhead
Accumulator	$2*N*N^2 * M$	N^2	xM
Counter	$C*N^2 * *M$	N^2	xM
Two multiplexers	$2*M*N^2$	0	$+2*M* N^2$
de-multiplexer	$M*N^2$	0	$+M* N^2$
Buffers		0	$+2*(N-1)^2$
Zero detecting logic*	-	-	-

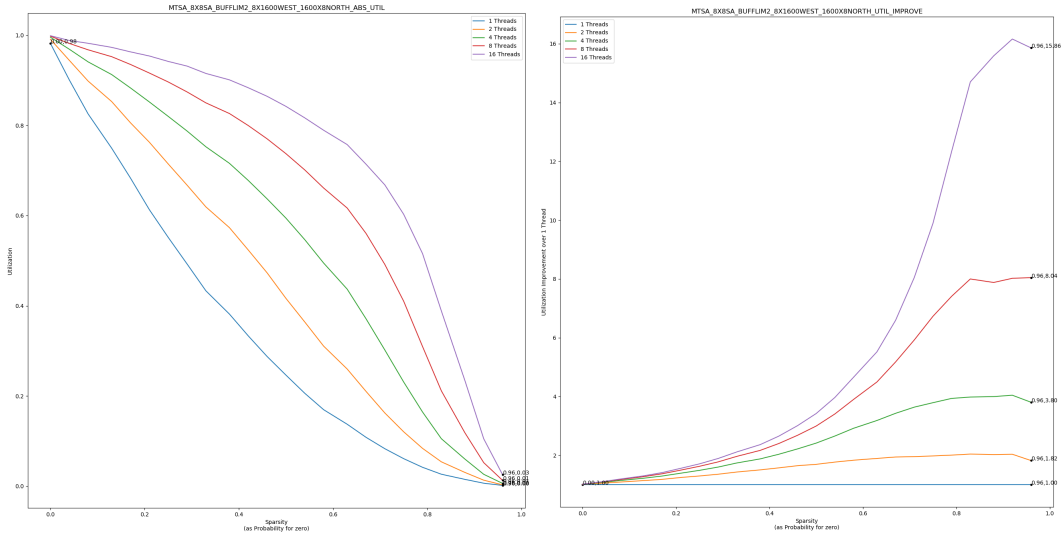

Fig. 14: MTSA utilization and utilization improvement over sparsity for different number of threads, with limited buffer depth of 2. Same as the unlimited buffer depth case, we fed the system with long matrices: 8X1600 from the west, and 1600X8 from the north.

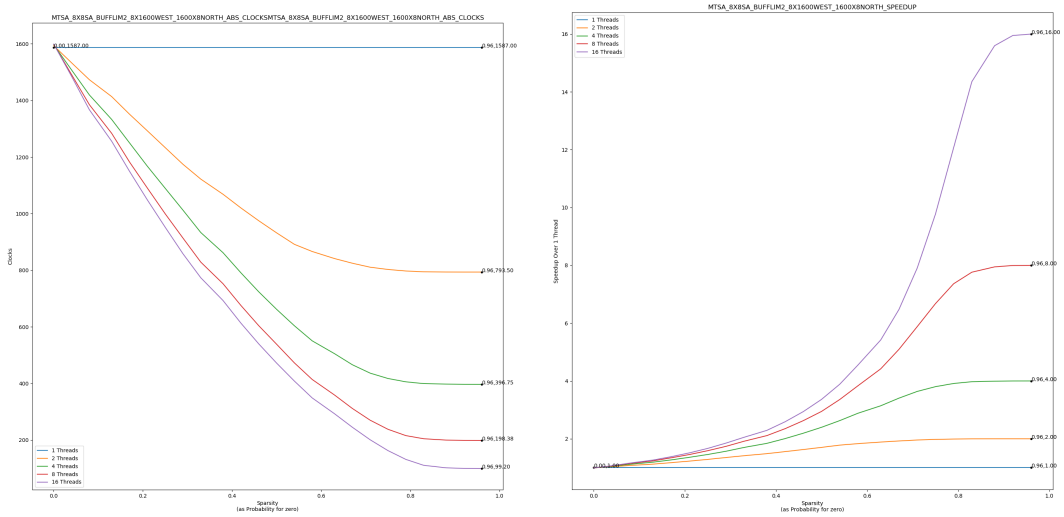

Fig. 15: MTSA clock cycles and speedup over sparsity for different number of threads, with limited buffer depth of 2

VII. CONCLUSION

We have shown that the MTSA system solve many of the traditional systolic array architecture problems:

- The MTSA keeps high values of utilization even in sparse environment (figures 8,9).
- The MTSA throughput increases in sparse environment as Von-Neumann architecture, and in opposite to the traditional systolic array architecture (figures 11,12,15).
- The MTSA method hide the inefficiency of the rigid structure of the systolic array in sparse environment.
- Utilization improvement and speed up correlates for every sparsity (figures 9, 12).
- Limitation constraints on the buffers depth cause relative minor influence on the MTSA performances.

We found that every sparsity value determines operating point which is optimal in terms of efficiency derived from the number of threads and its hardware overhead meaning. Dynamic change of the operating point of the system according to the dynamic changing of the sparsity might create the optimal throughput-overhead ratio.

REFERENCES

[1] Jouppi, Norman P., et al. "In-datacenter performance analysis of a tensor processing unit." 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2017.
[2] Kung, H. T., and Charles E. Leiserson. "Systolic arrays (for VLSI)." Sparse Matrix Proceedings 1978. Vol. 1. Society for industrial and applied mathematics, 1979.
[3] http://web.cecs.pdx.edu̇perkows/temp/May22/0020.Matrix-multiplicationsystolic.pdf
[4] http://www.telesens.co/2018/07/30/systolic-architectures/
[5] Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. Kim Hazelwood, Sarah Bird at all, 2018.
[6] Tang, B., Information, S. O., & University, Y. N. (2018).: Case study of the application of field programmable gate array fpga in the smart skill. Application of IC.
[7] Sparsity in Deep Neural Networks - An Empirical Investigation with TensorQuant. Dominik Marek Loroch, 2018.